(2)

AD-A201 963

NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

A SURVEY OF AUTOMATIC CODE GENERATING SOFTWARE

by

Sherman L. O'Brien

September 1988

Thesis Advisor: Daniel R. Dolk

89 1 04 014

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School; 6b OFFICE SYMBOL (If applicable) 54	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000	7b. ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)

A SURVEY OF AUTOMATIC CODE GENERATING SOFTWARE

12. PERSONAL AUTHOR(S) O'Brien, Sherman L.

13a. TYPE OF REPORT	13b TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Master's Thesis	FROM ______ TO ______	1988 September	81

16. SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17. COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Automatic code generation
			Automatic programming
			Computer languages

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The advances made in computer hardware development have long outdistanced the computer software needed to make that hardware perform useful work for the user. This has precipitated a software crisis in the industry and spawned many potential solutions for alleviating the crisis. Among the various solutions are software systems that will automatically write program code. This thesis examines four such software systems currently available to a system developer giving a brief description of the product, principle behind its operation and possible applications. Additionally, it provides the reader background information on computer programming languages, reasons for the software crisis, the software development life cycle, and a method of classification and taxonomy of software development tools. The thesis concludes that these tools, properly applied, can be useful in relieving the software crisis in an organization but will not eliminate the crisis

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a NAME OF RESPONSIBLE INDIVIDUAL Daniel R. Dolk	22b TELEPHONE (Include Area Code) (408)646-2260; 22c OFFICE SYMBOL 54Dk

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

Block 18 continued: Software crisis; Software Development Life Cycle (SDLC); Software development tools; Taxonomy of software tools

Block 19 continued: or the need for programmers.

A Survey of Automatic Code Generating Software

by

Sherman L. O'Brien
Lieutenant Commander, United States Navy
B.S., Iowa State University, 1974

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN INFORMATION SYSTEMS

from the

NAVAL POSTGRADUATE SCHOOL
September 1988

Author: Sherman L. O'Brien
Sherman L. O'Brien

Approved by: Daniel R. Dolk
Daniel R. Dolk, Thesis Advisor

Ronald E. Rautenberg
Ronald E. Rautenberg, Second Reader

[illegible]
David R. Whipple, Chairman
Department of Administrative Sciences

Kneale T. Marshall
Kneale T. Marshall,
Dean of Information and Policy Sciences

ABSTRACT

The advances made in computer hardware development have long outdistanced the computer software needed to make that hardware perform useful work for the user. This has precipitated a software crisis in the industry and spawned many potential solutions for alleviating the crisis. Among the various solutions are software systems that will automatically write program code.

This thesis examines four such software systems currently available to a system developer giving a brief description of the product, principle behind its operation and possible applications. Additionally, it provides the reader background information on computer programming languages, reasons for the software crisis, the software development life cycle, and a method of classification and taxonomy of software development tools. The thesis concludes that these tools, properly applied, can be useful in relieving the software crisis in an organization but will not eliminate the crisis or the need for programmers. [illegible]

TABLE OF CONTENTS

I. INTRODUCTION ---------------------------------- 1

A. THE ROLE OF SOFTWARE ---------------------- 1

B. PURPOSE OF THE THESIS ---------------------- 2

C. SCOPE OF THE THESIS ------------------------ 3

D. ORGANIZATION OF THE THESIS ---------------- 3

II. KEY SOFTWARE ISSUES --------------------------- 5

A. INTRODUCTION ------------------------------ 5

B. THE SOFTWARE DEVELOPMENT LIFE CYCLE (SDLC) -- 5

C. THE SOFTWARE "CRISIS" ---------------------- 9

D. COMPUTER LANGUAGES ------------------------ 12

E. AUTOMATIC CODE GENERATION ----------------- 19

F. SUMMARY ------------------------------------ 20

III. CLASSIFICATION AND TAXONOMY OF SOFTWARE DEVELOPMENT TOOLS ------------------------------ 22

A. INTRODUCTION ------------------------------ 22

B. GENERAL CLASSES OF SOFTWARE TOOLS ----------- 22

C. TAXONOMY OF SOFTWARE TOOLS ----------------- 23

D. SUMMARY ------------------------------------ 30

IV. REVIEW OF AUTOMATIC CODE GENERATING SYSTEMS ----- 32

A. INTRODUCTION ------------------------------ 32

B. USE.IT ------------------------------------- 33

C. COGEN ------------------------------------- 40

D. INFORMIX/4GL ------------------------------ 45

E. GENIFER ----------------------------------- 49

F. FEATURES OF SYSTEMS ------------------------ 55

G. SUMMARY ------------------------------------ 56

V. CONCLUSIONS AND RECOMMENDATIONS ---------------- 59

A. INTRODUCTION ------------------------------- 59

B. CONCLUSIONS -------------------------------- 59

C. RECOMMENDATIONS ---------------------------- 62

APPENDIX: AUTOMATIC CODE GENERATING SYSTEMS ---------- 64

LIST OF REFERENCES ------------------------------------ 70

INITIAL DISTRIBUTION LIST ---------------------------- 72

LIST OF TABLES

4.1 SYSTEM FEATURES ---------------------------------- 56

LIST OF FIGURES

2.1 Historical Life Cycle Model ---------------------- 6

2.2 Programming Language Abstraction ----------------- 18

2.3 Historical User--Programmer--Application Link ---- 20

3.1 Software Tool Taxonomy --------------------------- 24

4.1 Functional Life Cycle Process -------------------- 33

4.2 Hierarchical Control Map ------------------------- 35

4.3 Nodal Relationships ------------------------------ 35

4.4 USE.IT Control Map Example ----------------------- 37

4.5 Ring Menu -- 47

4.6 MAIN Menu Routine -------------------------------- 47

4.7 GENIFER Main Menu -------------------------------- 50

5.1 Programmer, User, Software Perspective ----------- 61

I. INTRODUCTION

A. THE ROLE OF SOFTWARE

Until a computer is given a set of explicit instructions telling it exactly what to do and in what order, the user will not receive any of its potential benefits. Contemporary computer users know the single device that releases its potential power is software.

Software is what makes the computer perform useful work. It includes but is not limited to the data, algorithms, and programming code used to tell the computer what to do. In today's marketplace many users are familiar with the standard floppy disk used by microcomputers to store programs and data. In larger computer systems the programs may be on tape or some other storage medium. Dedicated computers such as those used in a fire control system aboard a warship may have their software written directly onto a silicon chip--something more commonly associated with hardware. Regardless of the medium used to introduce the program into the computer it can all be termed software.

The overall importance of software in the computer industry is difficult to underestimate. A readily understandable measure is money. Software is big business and indications are that it has surpassed hardware as the most expensive part of a contemporary computer system. Over 10 years ago estimates for the World Wide Military Command and Control System (WWMCCS) were put at $50 to $100 million for hardware and $722 million for software [Ref. 1:p. 41]. More recently the Strategic Defense Initiative (SDI) has been deemed impossible because development of the required software is considered impossible [Ref. 2:p. 46]. Developing software involves the following steps:

1. Problem definition;
2. Software design;
3. Coding;
4. Testing;
5. Implementation;
6. Maintenance.

While some software projects may be larger than others these steps are involved in finding and developing a computer solution to a problem.

B. PURPOSE OF THE THESIS

Vendors claim that part if not all of the software development process can be automated with products they have introduced to the marketplace. This thesis will attempt to explore many of these products and answer the following questions.

1. What part of the development process does the product automate and what exactly does automate mean in the particular application?

2. To what degree does the product surveyed meet the need for a programmer interface between the user and the application?

3. Do products that claim to produce code automatically in fact write error free, syntactically correct code? How "automatic" is automatic code generation software?

4. If the answer to 3 is yes, what input is required to produce a computer program? What techniques are used to generate code?

C. SCOPE OF THE THESIS

This research project is limited to a static evaluation of commercially available software tools. Primary emphasis will be placed on products that generate computer code such as FORTRAN, PASCAL, or COBOL. Also included are fourth generation language application generators and microcomputer application program generators. Specifically excluded from the research project are compilers.

D. ORGANIZATION OF THE THESIS

The presentation of this thesis is organized into five chapters including this introduction. In Chapter II a number of key issues associated with software development will be presented including the Software Development Life Cycle (SDLC), the Software Crisis, and a brief look at the evolution of computer programming languages. The third chapter is a presentation of a classification and taxonomy of software development tools that was published by Raymond C. Houghton. The fourth chapter is a review of commercially available automatic code generating software systems. The final chapter contains conclusions and recommendations based on this study with suggestions for areas of further study related to this topic. The Appendix is a listing of other software systems that fit the automatic code generating classification.

II. KEY SOFTWARE ISSUES

A. INTRODUCTION

The introduction highlighted the important role software plays in a computer system. This chapter provides a more in-depth look at software and some of the key issues involved in its development. It will describe the software development life cycle, identify the stages of the life cycle that are the most likely candidates for automation, define the "software crisis," and offer reasons for the existence of the crisis. Finally, it presents a brief look at the problem of communicating with a computer in a language it understands and some of the solutions currently in use.

B. THE SOFTWARE DEVELOPMENT LIFE CYCLE (SDLC)

When viewing a problem for which a computer solution is being considered or has been found it is common to think in terms of a system, a collection of components assembled to interact and achieve some goal. In a computer system the components are computer equipment or hardware, programs, data, procedures, and personnel. [Ref. 3:p. 25]. The program component is often looked upon as a system itself. The development of the program, or software, system is known as the software development life cycle. The historical life cycle model (Fig. 2.1) is the view most often taken in the

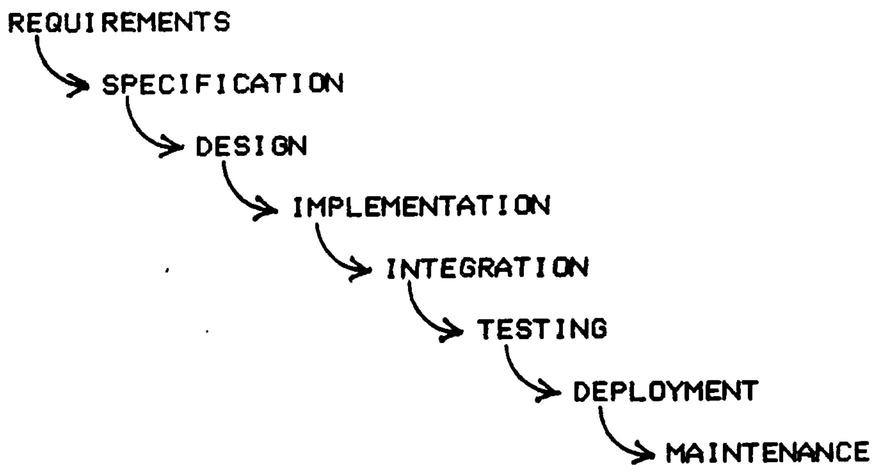

Figure 2.1 Historical Life Cycle Model

development of a software system. It is this life cycle model, or a variation thereof, that is the candidate for overhaul in the software development process. Automating all or part of the process can reduce errors, speed development, and reduce the cost. [Ref. 4:p. 26].

In the introduction to this thesis software development was presented as a six step process. As presented they are simply a variation of the historical life cycle process:

1. Problem definition;
2. Software design;
3. Coding;
4. Testing;
5. Implementation;
6. Maintenance.

All of these activities are labor intensive, costly, and prone to error. The prospect of automating them is exciting but may not be feasible.

The definition of the problem itself seems an unlikely candidate for automation. Technology in use today still requires that the user identify the problems that are candidates for a computer solution whether it be a business payroll system, an inventory system, or a game.

Once the problem has been identified, a method of solving it using a computer must be developed. Given that a computer solution must input certain data elements, manipulate the data, and return a solution identifies it as a known, structured activity. As such it can be looked upon as a candidate for automation. [Ref. 5:p. 1].

Coding in a software system is the act of putting the problem solution into a form understandable by the computer. It too is a structured task. In fact many of the languages used to program a computer are known as structured languages referring to the strong procedural and data structuring capabilities.

After a computer program has been coded, it must be thoroughly tested to validate the design and coding of the program. The importance of testing in the software development life cycle cannot be overemphasized. Estimates are that as much as 40% of the entire software development effort is expended on testing [Ref. 6:p. 89]. Because of the intense resource expense involved in testing, any automation of testing can prove of great value. Many categories of automated tools exist including test data generators and test file generators [Ref. 6:p. 316]. Automated testing tools will not be examined in this thesis unless the automated tools under examination include some automated testing features.

Implementation of the software system refers to the act of putting the system into regular use. The methods of implementation can vary as can the impact of the system on the users. The actual motion of putting the software in the computer and getting it to run will not usually involve the user except in microcomputer based systems or user developed systems. Automation therefore will likely be limited. The inclusion of training in the implementation step opens possibilities of on-line tutorials and computer-aided instruction as examples of automation.

The maintenance of software refers to the act of making some sort of alteration to existing software. This action takes different forms. Corrective maintenance fixes errors in the software while adaptive maintenance refers to modifications made so the software will properly interface with a changing environment. Recommendations for new features or modifications of existing functions comprises perfective maintenance. A rarer form of maintenance is preventive maintenance that is performed to enhance future maintainability or reliability. Research indicates that the major proportion of the maintenance effort is devoted to perfecting existing software. [Ref. 6:pp. 323-326]. It can be seen that the automation of software maintenance is dependent on the type of maintenance being performed. Rewriting code as a result of a change to factors in the algorithm, say a different method of figuring a sales commission, is similar to automating code generation. On the other hand, the design of an enhancement feature may be less adaptive to automation.

C. THE SOFTWARE "CRISIS"

In the evolution of the computer industry software has lagged behind hardware to the point that the problem is often referred to as a crisis [Ref. 6:p. 22]. It is not that the software available does not do what is intended but refers instead to the development and maintenance of software to meet a rapidly escalating demand from the user community.

The evidence to support the allegation that software is indeed in a crisis situation is ample. An early example of the difficulties involved with software development is found in a project that goes back over 20 years. The development of the IBM Operating System 360 fell far behind schedule in development and is a classic example in the software development literature of the problems associated with software project development [Ref. 7:p. 35].

More common in the current literature though are the backlogs of applications development. Carl Flood of Trans World Airlines states that the applications backlog at his organization is "tremendous" [Ref. 8:p. 651]. Other authorities simply state that there are backlogs of several years [Ref. 9:p. 60]. The applications backlog that is documented reveals only a part of the overall backlog. Frustrated users may look at the backlog as impossible to overcome and as a result stop making requests of the data processing shop of an organization. This invisible backlog may rival the documented backlog in size. [Ref. 10:p. 4].

There are many reasons for the software crisis. For one thing, after software is developed it requires maintenance. The force of personnel needed for maintenance is the same as the one used in the initial development. As a consequence, the size of the force available for further software development decreases as more applications are developed.

A dramatic increase in the amount of potential computing power in the hands of users has also contributed to the crisis. One measure of computing power that can be used is the amount of main memory available to the user. In 1981 International Business Machines (IBM) offered their personal computer (PC) with 64 kilobytes of main memory at a retail price of about $3000.00. In contrast, the Atari Corporation recently introduced their 1040ST model PC with one megabyte of main memory at a suggested retail price of $999.00.

[Ref. 11:p. 87]. Similarly, Digital Equipment Corporation's MicroVAX computer provides power similar to that of the larger VAX line of computers, but at about $30,000 is only a sixth the price of the larger machine [Ref. 12:p. 44].

In this price range the user community widens to include individuals and smaller business entities. While this trend may be hailed by some, it further complicates the software crisis by diversifying the application demand. The traditional scientific applications are joined by a stockbroker's client tracking system and dispatching and billing systems for transportation companies. [Ref. 12:p. 44].

Another factor contributing to the software crisis is the overall shortage of development personnel. With the increase in demand from the aforementioned new user community, many companies that previously maintained an in house development team are finding it no longer economically possible. Experienced software developers are lured away to more lucrative jobs with software development firms that are dedicated to the development of software for a larger group of users.

There is no simple solution to the software crisis, because the causes are so diverse. Among the many ingredients that could possibly help the software crisis are better and more diverse software development tools. The remainder of this thesis will focus on various software development tools currently available from commercial vendors.

D. COMPUTER LANGUAGES

Unleashing the potential problem solving power of a computer requires that the computer be told what to do in a manner that it understands. In this section, a brief review of the languages that have been developed to communicate with a computer will be presented. The review will include a brief description of the language, its uses, strengths and weaknesses.

Because of the overwhelming number of languages that have evolved in the short history of digital computers it would be impossible to present them all here. In fact, it is realistic to say that a comprehensive presentation of all computer languages ever developed could never be made, simply because some are developed for one time special applications or solely for research purposes. As such they may never come to the attention of the community at large. The languages chosen for presentation here were selected because they represent a major milestone in language development, are in such wide use throughout the software development community that they represent a significant percentage of the overall market, or have some other attribute that makes it significant to this study.

At the heart of every computer is the machine language. The language is used by the programmer to communicate with the computer at the most basic hardware unit, the logic gate. It is characterized by binary code, strings of 0's and 1's, and any program written in machine language is referred to as an object program [Ref. 13:p. 145]. The strength of machine language is that well written and documented programs can make efficient use of a computer's main memory and optimize execution of the program. The advantages of machine language are offset by the many disadvantages, however. In an environment where labor resources are already used intensively, machine language programming only adds to the problem. The resultant code is difficult to read, test, and maintain. Because the programmer specifies exactly what operation is to take place and the address of the data to be operated on, the simple act of inserting or removing a single line of the program will result in the addresses of all other instructions to be incorrect. Finally, machine language programs are hardware specific and portable only to other processors that share exactly the same instruction set. [Ref. 4:p. 11]. Currently machine language is seldom used and for most applications would be inappropriate.

Closely associated with the machine language of a particular processor is the assembly language. The most noticeable characteristic of assembly language is its mnemonic form. Instead of working with binary strings the programmer writes the program using shorthand style words for each instruction in the program. The other advantages and disadvantages associated with machine language also apply to assembly language. Assembly language is still used for writing applications; at Trans World Airlines, for instance, most applications are still written in assembly language because of a need for processing efficiency [Ref. 8:p. 657].

There is a fine line between what has previously been cited as machine language and assembly language. The merits of the debate put forth by the various factions is not germane to this thesis. What the reader must remember is that any other symbolic or artificial means of programming a computer must eventually be translated into this machine understandable form. The methods in use are compilers and interpreters. These are also software components but the difference between them is subtle. An interpreter will read a program statement-by-statement, convert it to machine code, and executing the statement "on-the-fly." A compiler on the other hand will read the entire program before it is executed and can thus realize efficiency in execution speed and memory optimization.

The first artificial language to be developed and gain widespread use is Formula Translating (FORTRAN) System that was developed by an IBM research group headed by John Backus. The language specifications were first published in 1956 and pertained only to the IBM Model 704 computer. From this humble beginning, the language grew to become what is widely recognized as the most popular programming language among the engineering and scientific communities. Some of the criticisms of FORTRAN were that it lacked direct support of the structured constructs, had poor data typing, and could not easily support string handling. Some of these deficiencies have been corrected in newer standards of the language.

Perhaps the most widely implemented language is the Common Business Oriented Language (COBOL). As the name implies this language is oriented towards business data processing applications. COBOL was developed for applications with relatively simple algorithms, but a high degree of input and output such as a payroll or inventory for instance. COBOL is also characterized by its English-like syntax that makes the statements somewhat easier to read and understand. Though the syntax makes COBOL programs fairly easy to write, even the simplest of programs can become quite lengthy.

A common first language for beginning programmers, and one that is usually available on all microcomputers, is BASIC, the Beginner's All Purpose Symbolic Instruction Code. It was developed at Dartmouth College in 1965 as an easy to learn, easy to use first language that could serve as a stepping stone to the more powerful languages such as FORTRAN. The language constituents, LET, RETURN, IF, etc., are easy to understand, but many versions of the language have developed over the years and fluency in the language by a user still requires dedicated study.

A language that has gained prominence recently, particularly in Department of Defense arenas, is ADA. Named for Lady Ada Lovelace, it was developed by the Department of Defense as the standard language for embedded computers, i.e., computers that are integral parts of larger systems. It imitates other languages in structure and notation, but supports many features to assist in interrupt handling, multitasking, and operations at a machine dependent level. Because ADA is required for use in many Department of Defense applications, it can be assured of a strong future. An important aspect of ADA is that strong measures are being taken to insure standardization. The Department of Defense registered ADA as a trademark and will not permit use of the name on a compiler unless it has been validated by them.

Finally, the current vogue in computer languages is what are termed fourth generation languages, 4GLs for short. The first generation of languages was machine code, the second generation assembly language, and the third generation those machine independent languages such as FORTRAN, COBOL, BASIC, and others. An exact definition of 4GLs is difficult. Some would call them data base languages but not all use a data base. Others call them nonprocedural languages, but many contain a procedural code. James Martin uses the following benchmark to define 4GLs: if a language cannot produce results for its users in one-tenth the time required using COBOL it should not be called a fourth generation language. [Ref. 10:p. 28]. Whether it is a data base language or nonprocedural language is really not important. The distinguishing feature of a 4GL is that it allows the user to specify the desired solution and not break the problem into an algorithm. Their strength is in the English-like statements that provide fast, efficient development of applications using high-level, nonprocedural specification syntax. [Ref. 9:pp. 48-49]. Commercial examples of 4GLs are IBM's Structured Query Language (SQL) and Cincom's MANTIS. In the microcomputer inventory of software Ashton-Tate's dBASE relational database software could be classified a 4GL. There are no standards yet for these languages, although SQL is currently under consideration.

As each generation of computer language has evolved, the level of abstraction associated with it has increased. Figure 2.2 shows the relationship between the level of abstraction and the generation of the programming language. On the vertical axis the value LOW corresponds to the binary code of 0's and 1's while HIGH equates to the vernacular. It is emphasized that this evolution is not a continuous function and the exact slope of the curve is likely to be a matter of debate. However, the relative position of the points is accurate. The fourth generation languages with their strong English-like statements are further from the binary machine code that the computer understands than any other language to date. This level of abstraction and sophistication means that the user, more than ever, is reliant on compilers to carry out the detailed translation.

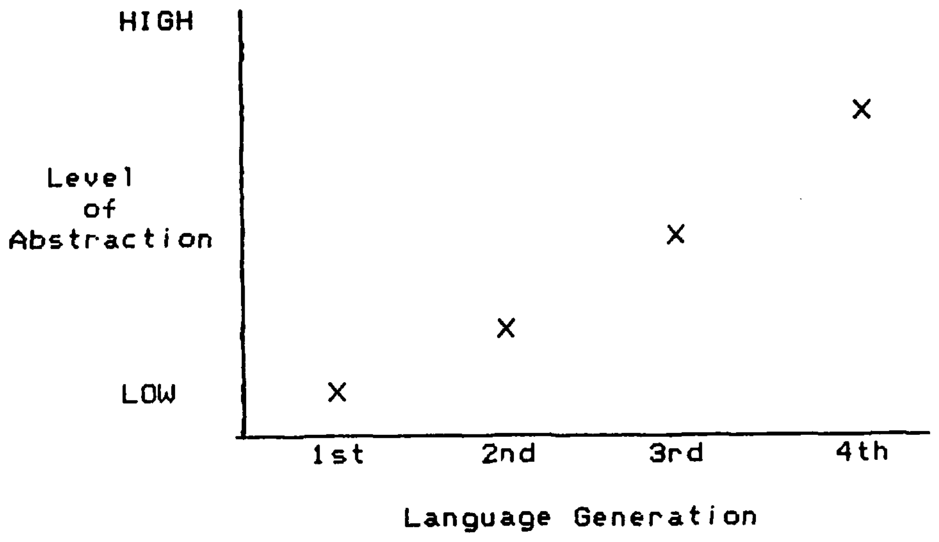

Figure 2.2 Programming Language Abstraction

Generally it can be said that successive generations of software serve as "automatic" code writers for the preceding generation. Development of each language is usually an attempt to provide the user with a more powerful method of communicating instructions to the computer. However, because each language brings with it a particular syntax and lexicon the common errors involved with coding a computer program remain. Full realization of the potential power of a language requires extensive training on the part of the user. Still, even among the most knowledgeable of programmers errors continue to occur and software development costs continue to rise because of the interface between the human code writer and the exact requirements of the language.

E. AUTOMATIC CODE GENERATION

In the preceding section describing the SDLC, each of the stages was assessed regarding its candidacy for automation. Many developers have chosen to automate the task of coding. Utilizing an automatic code generator in the software development process can attack some of the causes associated with the software crisis. First, because of the precise manner in which computers perform a task, many of the errors related to syntax and vocabulary can be eliminated. Although the debugging process of software development involves much more than eliminating errors of this type, the error reduction of any type is most cost effective the earlier it is accomplished. In addition, it is probably safe to assume that a computer can write code at a rate that exceeds that of even the most proficient coder.

A second important point to consider in the automatic generation of code is the possibility of eliminating the programmer as the link between the user and his application. The historical SDLC and its derivatives imply the use of a programmer/analyst to guide the user's specification to an application as depicted in Figure 2.3. The user's level of participation in the process is variable and often is a cause for many disgruntled customer's complaints that what was received was not what they specified. With the user in a position to utilize a tool that can automatically put a specification into code, such problems could be avoided. It is likely too that the application backlog would decline and an already austere supply of programmer/analysts would be available for newer applications.

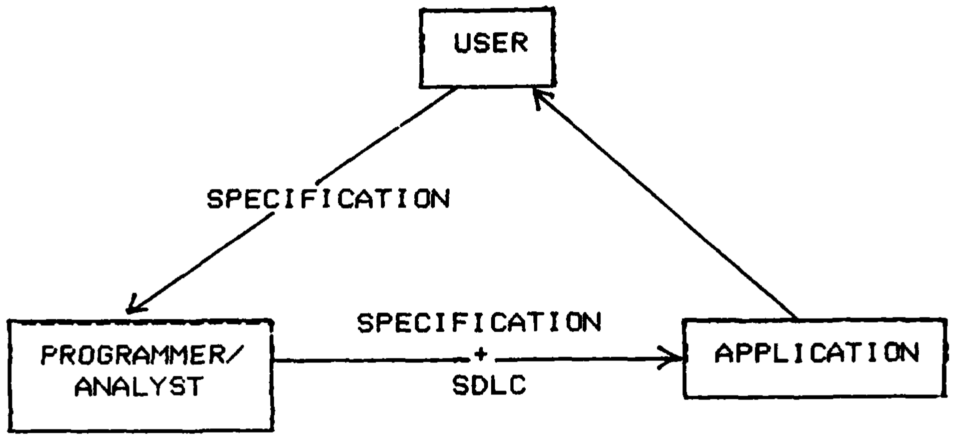

Figure 2.3 Historical User--Programmer--Application Link

F. SUMMARY

This chapter has focused on some of the many issues involved with the development of software. It has reviewed the Software Development Life Cycle (SDLC) and assessed the likelihood of automating the different stages. A look at the software crisis followed where causes were identified and possible solutions presented. The third section reviewed the way man communicates with the computer, through computer languages. A short review of the evolution of the languages from first generation machine code to the more recent fourth generation languages was presented. Finally, the automation of the code writing stage of the SDLC was presented along with potential areas of savings in the development of software.

In the next chapter a taxonomical view of commercially available automatic code generating software will be presented. In addition, specific products will be reviewed and classified.

III. CLASSIFICATION AND TAXONOMY OF SOFTWARE DEVELOPMENT TOOLS

A. INTRODUCTION

The previous chapter presented a number of issues related to the development of software systems and introduced software systems that automatically generate the code required by the system. This chapter will make a general classification of software development tools and follow with a detailed breakdown of the features that may be found among them. The primary source for the information in this chapter is from work by Raymond C. Houghton, Jr. for the National Bureau of Standards (NBS). It is published in NBS Special Publication 500-88, "Software Development Tools." The study presents a solid base for software development tool classification and taxonomy. However, because it was issued in 1982 after 3 years of study, many of the currently available tools, particularly the 4GLs, were not available at that time and are not included in the study. Where necessary to accommodate this new technology, Houghton's work will be appropriately expanded.

B. GENERAL CLASSES OF SOFTWARE TOOLS

It has already been shown that automation of every stage of the SDLC has been attempted. The earliest and most common to most users are the compilers, debuggers, and editors. Currently the inventory includes program generators, application generators, and software systems design development tools. Houghton put these tools into a classification scheme of six categories. They are:

1. Software Management, Control, and Maintenance Tools;
2. Software Modeling and Simulation Tools;
3. Requirements/Design Specification and Analysis Tools;
4. Program Construction and Generation Tools;
5. Source Program Analysis and Testing Tools;
6. Software Support System/Programming Environment Tools.

In Houghton's work, two-thirds of the tools studied were from categories (1) and (5). One of the less populated environments he noted was category (4). The tools studied in this thesis all belong in this class.

C. TAXONOMY OF SOFTWARE TOOLS

To provide users with a more useful method of determining tools of interest, a taxonomy of their features was developed by Houghton. Each of the tools studied was then classified by features. Classification is a hierarchical arrangement (Figure 3.1) with the highest level being the most abstract and including all features. The second level covers the basic processes of a tool; input, function, and output. Note that these are the basic functions common to any system and particularly a software system. At the third level of the taxonomy are the classes of tool features. They are:

1. Subject (I);
2. Control input (C);
3. Transformation (T);
4. Static analysis (S);
5. Dynamic analysis (D);
6. User output (U);
7. Machine output (M).

From this level Houghton identified a total of 53 tool features that are at the bottom of the hierarchy.

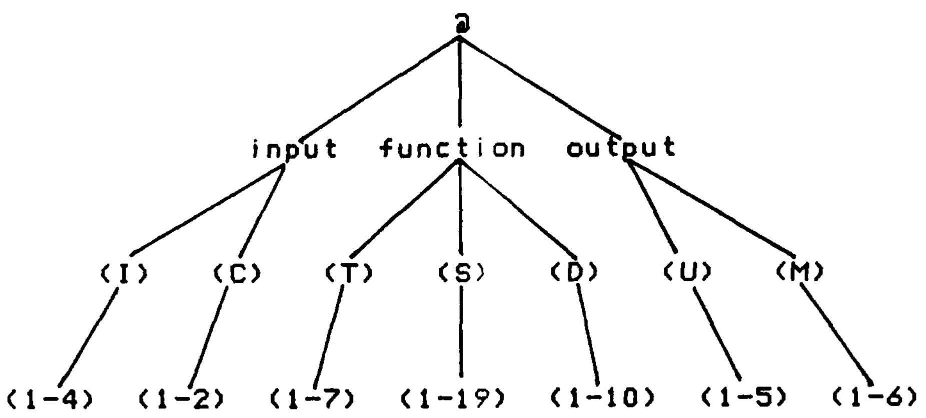

Figure 3.1 Software Tool Taxonomy

The following paragraphs will detail this taxonomy of tool features, with definitions of most of the terms used in this classification.

1. Input

The forms of input to the software tool fall into two classes. Control input (C) is defined as how the tool should operate, and subject (I) is based on what the tool should operate on. The latter of these is usually the main input to the tool, and according to Houghton has four types: code, very high level language (VHLL), data, and text. Code is a high level, assembly, or object level language while VHLL, according to Houghton, refers to languages that are not in an executable form. Many of the systems that use VHLL input are oriented to specifications, descriptions, or requirements. Of the 37 automatic code generating software systems catalogued by Houghton, only one did not use either code or VHLL as subject input.

2. Function

Input to a software system is processed by functions that fall into three classes: transformation, static analysis, or dynamic analysis. Of the 53 tool features listed in Houghton's study, over two-thirds are subsets of this basic tool process. Among the automatic code generating systems the most common are the formatting, translation, and synthesis features in the transformation class. Very few of the automatic code generating systems utilize the features found in the static and dynamic analysis classes.

As noted, formatting, translation, and synthesis are the most common function features of the automatic code generator systems. Formatting refers to the arranging of a program according to predefined or user defined conventions. Examples seen in tools using this feature are alphabetized variable declarations and indenting statements. The translation function is the conversion from one language to another. Tools that use the synthesis feature are generating an application or program from a specification or from an intermediate language. Houghton identified this feature as one that was found in application and program generators, and that much promise for increased programmer productivity was associated with these tools. Restructuring, optimization, editing, and instrumentation are the other features in the subset of the transformation function.

Static analysis features describe the operations on the subject with no regard to the executability of the subject. Houghton identified 19 features in this class. Of the tools in the automatic code generating class, only 21% display any static analysis features; of those that do, management and error checking are the most common. The many features of static analysis and their definitions are:

1. Management--aiding the control of software development.

2. Cross reference--logical reference of entities to other entities.

3. Scanning--sequential examination of an entity to identify key areas or structure.

4. Auditing--an examination to determine adherence to predefined rules.

5. Data flow analysis--graphical analysis of the sequential patterns of definitions and references of data.

6. Consistency checking--determines if each entity is internally consistent with uniform notation and terminology, and is consistent with its specification.

7. Statistical analysis--performs data collection and analysis for statistical purposes.

8. Error checking--determination of discrepancies, their importance and/or cause.

9. Structure checking--detection of structural flaws within a program.

10. Comparison--determining and assessing the differences between two or more items.

11. Completeness checking--determining if an entity has all its parts present and if its parts are fully developed.

12. Complexity measurement--determining how complicated an entity is by evaluating some number of associated characteristics.

13. Tracking--following the development of an entity through the life cycle.

14. Interface analysis--checking for consistency and adherence to predefined rules along the interfaces between program elements.

15. I/O specification analysis--analysis for the purpose of generating input data.

16. Type analysis--evaluating the domain of values attributed to an entity to ensure proper and consistent definition.

17. Cost estimation--assessing the behavior of variables that impact the life cycle cost.

18. Units analysis--determining if the units of physical dimension of an entity are properly defined and consistently used.

19. Scheduling--assess software development schedule and its impact on the life cycle.

The final features in the function class are those defined as dynamic analysis. These are operations that are determined during or after execution. It requires some form of symbolic or machine execution and provides information about the program's execution behavior. Houghton's study found only one system in the application and automatic code generation class that made use of any sort of dynamic analysis features. The 10 features in this class and their definition's are:

1. Coverage analysis--used to determine the adequacy of a test run by determining and assessing measures associated with the invocation of program structural elements.

2. Tracing--making a historical record of execution of a program.

3. Tuning--specifying which parts of a program are executed most often.

4. Simulation--computer generated representations of features of a system.

5. Timing--reporting the CPU times associated with the program.

6. Resource utilization--analysis of the use of hardware or software resources.

7. Symbolic execution--utilization of symbolic rather than actual data values to reconstruct logic and computations along a program path.

8. Assertion checking--checks user-embedded assertion statements between elements of a program.

9. Regression testing--detects errors that may have been caused by changes or corrections made during software development or maintenance, by rerunning test cases that have been previously properly executed.

10. Constraint evaluation--generates and/or solves input or output constraints to determine test input or to prove correctness of the programs.

3. Output

The links between the tools and both the user and the computer make up the output features. Features take on different types and different forms.

User output is the type that the tool returns to the human user. According to Houghton it falls into one of the five following types:

1. Listings--probably the most familiar form to most users, and lists the source programs or data.
2. Tables--exhibition of facts or relations in a definite, compact, and comprehensive form arranged in columns.
3. Diagnostics--machine output that indicates the discrepancies in a system.
4. Graphics--graphic presentation of operations, flow, etc., such as flow charts, hierarchical tree, and control maps.
5. User-oriented text--natural language (English) output, such as documentation and reports.

Application and program generating systems catalogued by Houghton incorporate all of these features, except user-oriented text, with two-thirds of the systems utilizing listings and diagnostics as their user output.

Machine output features are those that manage the interface between the tool and the computer or another non-human user, another tool perhaps. In Houghton's study six features are identified. They are:

1. Source code--a program written in a language that must be translated before execution; COBOL for instance.
2. Data--meaningful representations of characters or numeric values.
3. Object code--machine language output; normally the output of a translation process.
4. Intermediate code--code that is classified between source code and machine code.
5. VHLL--a program written in a very high level language.
6. Prompts--operators that interactively prompt the system operating the tool that it is ready for the next input.

The overwhelming machine output feature found in the application and code generating systems by Houghton is source code. None of the systems produced VHLL or prompts as their machine output.

The many features presented serve as one means by which software development tools can be evaluated by a potential user. Others that must also be considered are the environmental factors of portability, hardware requirements, and implementation language. And, of course, a tool must be available.

D. SUMMARY

This chapter has presented the reader with a detailed look at the many features identified by Raymond C. Houghton, Jr. in his study of software development tools. In addition, the features most commonly found among the tools identified as Program Construction and Generation Tools have been noted. Though the study is somewhat dated, particularly in computer technology terms, the taxonomy developed is sound and can be applied to the tools that are the subject of the following chapter.

The next chapter will review four software development tools that fit in the Program Construction and Generation Tools category and that are currently available in the commercial marketplace.

IV. REVIEW OF AUTOMATIC CODE GENERATING SYSTEMS

A. INTRODUCTION

Thus far this thesis has presented the problem of software development and how it has contributed to the software crisis. One possible remedy to the crisis is the automation of the SDLC or some parts thereof. Among the many automation tools available to software developers are those that automatically generate code. This chapter will review four commercially available products that fit into this category. The four systems are:

1. USE.IT from Higher Order Software, Inc.;
2. COGEN from Bytel Corporation;
3. INFORMIX/4GL by Relational Database Systems, Inc.;
4. GENIFER from Bytel Corporation.

Primary sources for the information presented are user documentation and library sources. Additionally, two users with USE.IT experience shared their thoughts. A demonstration and hands-on experience with GENIFER was also available.

The systems reviewed are by no means the only ones available nor are they presented here because they are either the best or worst examples. They do, however, represent two distinct subsets of Houghton's Program Construction and Generation Tools classification. USE.IT and COGEN both generate application program source code written in third generation languages, USE.IT in COBOL, FORTRAN, PASCAL, ADA, or C and COGEN in COBOL. INFORMIX/4GL and GENIFER are fourth generation language application development tools. The source code produced by GENIFER is the Ashton-Tate dBASE programming language. INFORMIX/4GL uses as its program source code Relational Database System's RDSQL.

B. USE.IT

Higher Order Software, Inc. introduced USE.IT in 1982 as an integrated systems development tool. Based on the functional life cycle process (Figure 4.1), USE.IT is an automated systems engineering tool that is incorporated by the analyst to design, analyze, and implement the desired system.

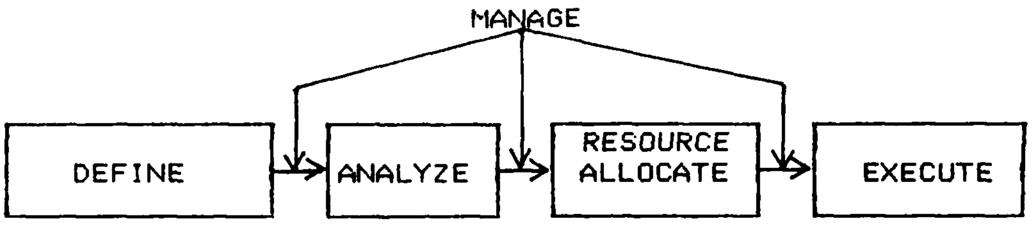

Figure 4.1 Functional Life Cycle Process

The functional life cycle model is defined by Hamilton and Zeldin as a formal model of the functions and the relationships between those functions which exist in a system for effectively developing a system. It could be a model for developing a software system, a hardware system, or some combination thereof. It can also include a system of people. [Ref. 4:p. 38].

USE.IT, in the role of manager in the functional life cycle model, automates the functional life cycle as follows. The system analyst translates user requirements into the USE.IT requirements definition language, AXES. Once the requirements have been defined, analysis of the requirements by the Analyzer component of USE.IT insures logical completeness and consistency. Finally, the Resource Allocation Tool (RAT) transforms the analyzed Axes specification to another representation. This is usually third generation language computer code but other options exist. A more detailed description of this process follows.

Specification of the system requirements is accomplished using a graphical representation of the functional breakdown of the system known as a hierarchical control map (Figure 4.2).

Decomposition continues from root node to the leaf nodes where further decomposition is not required. Every node of the control map is specified in terms of data types, function, and control structure (Figure 4.3).

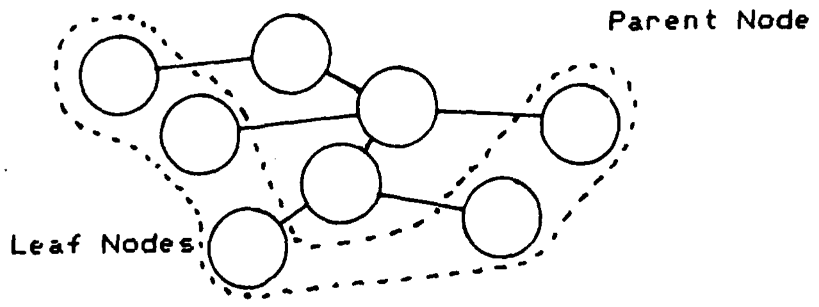

Figure 4.2 Hierarchical Control Map

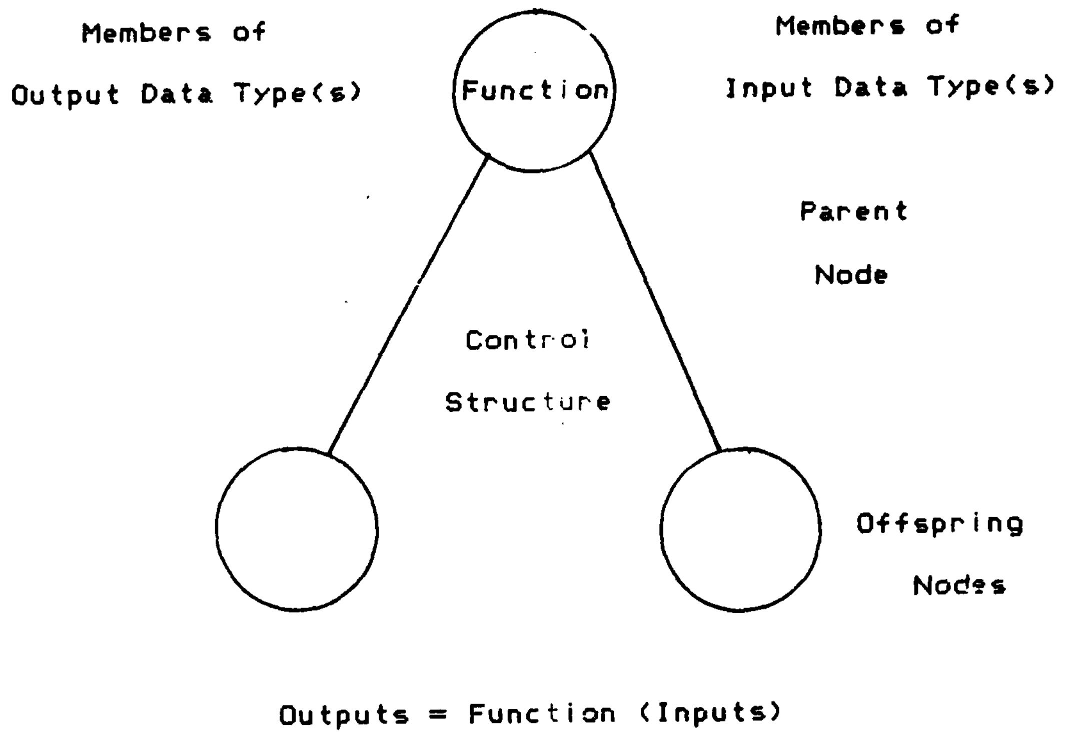

Figure 4.3 Nodal Relationships

Input and output values are individual data types and can be integers, rational numbers, Boolean values, or other data types. A function relates the input to the output. The control structure specified at each node defines the control relationship between a parent node and its offspring. In USE.IT technology there are three primitive control structures: Join (J), where offspring are processed in sequence, Include (I), where offspring are processed in parallel, and Or (O), where a choice must be made between offspring. There are more complex control structures that can be defined in terms of these primitive structures.

The leaf nodes fall into four categories: primitive operations, defined operations, external operations, or recursion. Primitive operations, labeled (P) on the control map, are the lowest level of decomposition. They are very low-level functions frequently performed. Common primitives can be collected in a system library. The defined operations, labeled (OP) on the control map, represent a call to a user-defined operation or subroutine. External operations, labeled (XO) on the USE.IT control map, represent calls to user-supplied subroutines not generated by other USE.IT specifications. Leaf nodes that are called recursively by a higher level node are labeled (R) on the control map.

A simple example of the USE.IT control map is presented in Figure 4.4. The control map is a decomposition of a

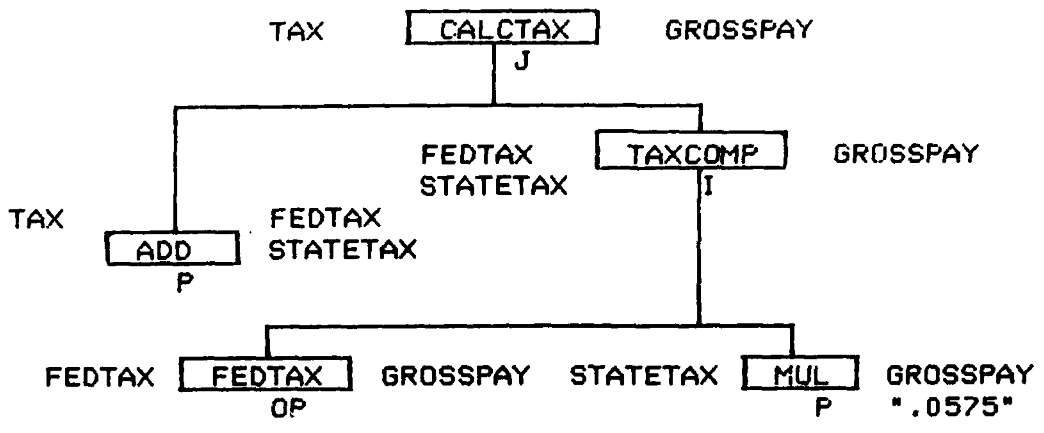

Figure 4.4 USE.IT Control Map Example

function that figures federal and state tax to be deducted from gross pay. In the example all inputs to the process and outputs from the function are defined at the top node. In the example the input is gross pay and the output is tax. In every nodal family the offspring receive inputs from the parent. The function is broken into two offspring functions, TAXCOMP and ADD. They are performed in sequence in a join control relationship. The TAXCOMP node is further decomposed into two other offspring nodes, MUL and FEDTAX, that operate in parallel using an include control relationship. Calculation of the STATETAX is a primitive function that multiplies the grosspay input by 0.0575 to output the tax. A defined operation, probably a table look-up, is used in the FEDTAX function to produce the output federal tax. Another control map would be needed to show the detailed decomposition of the FEDTAX function.

The user can call on the analyst at any time to perform an analysis of the specifications created with the graphic editor. A specification that is found to be logically complete and consistent is converted into the specification language AXES. The analyzer tests the specification for logical completeness by detecting missing functions or missing data and by guaranteeing that the hierarchical control map stops at primitive operations on defined data types. Consistency is insured by enforcing correct interfaces and data flow. Errors are corrected by the analyst with the graphic editor. Early incorporation of this feature allows the programmer/analyst to prototype the target application giving the end-user an early look at the application. Any needed modifications can then be made early in the development process.

An AXES specification that has been proven logically correct and consistent is passed to the Resource Allocation Tool. It is the function of the RAT to automatically generate either source code or object code for the target machine. The RAT accomplishes this by translating the Axes language specification into the target language by means of a general purpose translating algorithm. Detailed description of the algorithm is proprietary information presently unavailable. The source code that the RAT produces is guaranteed by USE.IT to be "bug-free." Specifications that are processed for one environment (e.g., FORTRAN) can be processed to another (e.g., COBOL). This feature gives management the flexibility of transferring a system from one environment to another. Potential errors in coding can thus easily be avoided. USE.IT does not allow the user to reverse engineer a system. Taking an existing program and generating a corresponding control map would give a programmer/analyst such options as verifying that the system is, in fact, logically complete and consistent, producing documentation for an undocumented system, or making modifications and insuring completeness and consistency.

The source code produced by the RAT can be manually modified. In general, however, it is the intention of USE.IT to perform all programming, including modifications and other maintenance, at the graphical level to maintain completeness and consistency.

Documentation is produced by USE.IT in the form of a documented hierarchy of requirements. The analyzer will produce documented error messages or report the fact that no errors exist if that is the case. Documented code from the RAT can be requested. The graphically represented system specification requires a plotter to produce.

USE.IT is available under the DEC VAX/VMS operating system and can generate COBOL, FORTRAN, ADA, and PASCAL source code. Under the IBM MVS operating system it constructs COBOL source code. A complete system, including

AXES generator, analyzer, and RAT (for one language only) is approximately $100,000.

C. COGEN

COGEN is an automated COBOL program generator developed and marketed by Bytel Corporation. COGEN's target for generation is standard COBOL business applications such as file maintenance, inquiries, and reports. COGEN works interactively with menus and data entry screens to produce files that are stored for use in independent programming or by COGEN to produce a source program. Code produced by COGEN can be modified if necessary and incorporated into other routines.

The series of main menus around which COGEN is organized includes five modules:

1. Data Dictionary;
2. Screen Processor;
3. Report Writer;
4. Batch Processor;
5. Program Generators.

Each of these main menu selections leads the user to a series of data entry screens that generate the modules, referred to as copy files, that are later used to generate applications.

A Data Dictionary selection by the programmer will result in program modules for file and record layouts. Data definitions are entered that describe the fields in the record layout. Special options and extensions can be included for data validation and default values. COGEN generates three COBOL files, FILE SELECT, FILE DESCRIPTIONS, and FILE DECLARATIVES, when data is defined. These files can then be included in any further program generation.

The Screen Processor selection produces files for screen input and output. These screens are then used for interactive input/output operations when COGEN-generated programs are executed. The first step taken by the programmer is to paint an image of the screen specifying the input/output fields. After the screen has been painted, COGEN presents the programmer with a series of questions that complete screen definition including validations and computations. The data dictionary is consulted for data definitions. A printer option allows a printout of the screen image together with field definitions. Files produced include the screen storage layout and input/output logic.

Production of printed reports comes from the files produced from the Report Writer. Report Writer uses one master file and multiple reference files to construct the program. Through a series of prompts to the programmer, COGEN defines headers, report layouts, conditional printing, control breaks, page size, reference file data selection, and so forth. After a report name is defined, the name and type of master file are designated. Report parameters can be printed using the print option. The report working storage and input/output logic files are produced with this option.

Processes that can be run in batch without operator intervention are created by the programmer/analyst with the Batch Processor. The copy files produced by the module access data files, select input records, perform computations, and update output data files. The copy files produced by the module are batch working storage and update logic.

Production of source code for business applications programs is the function of the Program Generators module. Creation of the application programs uses the copy files that are produced in the other modules. Programs that can be produced are Maintenance, Inquiry, Report, or Batch. After making the selection, the programmer must then name the program and designate the copy files and screens to be used. Independent copies of the previously created copy files are copied into the final program.

As an example, the tax calculation model illustrated in the previous section could be coded with COGEN. After making a thorough design of the application the programmer/ analyst would be ready to begin with COGEN. The first step is to specify data definitions such as GROSSPAY and TAX using the data dictionary module. In this module the programmer works in an interactive environment to describe each element with respect to level, type, and other appropriate COBOL language requirements. Default values and validation arguments can be specified when appropriate in this module. Depending on the user requirements, the process would be further developed for either a batch or interactive environment. An interactive environment would require screen design and field definition including required validations and computations. Data obtained while working the COGEN tutorial indicates that in the validation field at least two validation criteria could be specified. The format is an IF-THEN-ELSE type of statement. A determination of further nesting of arguments could not be made. A batch process would require the user to specify the input files to be read, computations to be performed and desired output.

In comparison to USE.IT, the COGEN process is strictly a code generator. It does not automate system definition or perform a completeness and consistency analysis. Therefore, the programmer/analyst retains the responsibility for system design. This evaluation was confirmed during a visit with Mr. Dan Pines, President of Bytel. Mr. Pines stated that COGEN is strictly a programmer's tool and is not intended to remove the programmer from the user-programmer-application loop. A user with at least an introductory course of study in COBOL would be able to use COGEN with success. In addition, while it is capable of producing applications in total, Mr. Pines affirmed that in some instances the programmer would be required to modify the 20% or so that COGEN could not produce. Above all else though, Mr. Pines ardently believes that the key to success in any programming endeavor is complete and proper data description. It is his belief that once a programmer has a complete, accurate design then COGEN will be of value in relieving the chore of writing code [Ref. 15]. Like USE.IT, COGEN produces "bug-free" code. The code can be modified as needed and incorporated into other routines.

In summary, COGEN is designed to improve programmer productivity. Through each section of the main menu the programmer/analyst specifies a library of files that can be assembled into an application program. By design, COGEN is a tool limited to the business application environment. USE.IT, on the other hand, is capable of a wider variety of applications making it more general in its design purpose. Because of the different design purposes of the two systems, determining which of the two systems, USE.IT or COGEN, is better cannot fairly be stated. A prospective user must evaluate needs against the potential solutions and make the appropriate decision.

COGEN runs on mini- and microcomputers under a variety of operating systems including VAX/VMS, CP/M and CP/M-86, UNIX, MS-DOS, and others. COBOL source code produced adheres to ANSI-74 COBOL standards. Fees for COGEN vary between \$950-\$7,500.

D. INFORMIX/4GL

Informix-4GL from Relational Database Systems, Inc. is designed as an application development package. It's intended to ease the programmer's job when it comes to creating menus or submenus, designing screens for data entry and retrieval, extracting information, and formatting reports. Relational Database Systems built Informix-4GL using their database management language, RDSQL. RDSQL is based on the Structured Query Language (SQL) developed by IBM. SQL is proposed as the ANSI standard for SQL implementations and RDSQL conforms to the draft standard.

A key feature in the description of a fourth generation language is that it must allow the programmer to specify the desired results without detailing the algorithm needed to accomplish the result. With Informix-4GL the programmer can accomplish a number of functions using a few brief statements. Among the functions that can be performed are:

1. Create menus;
2. Collect input from screen forms;
3. Use SQL to manipulate a database;
4. Call for help screens;
5. Create reports;
6. Collect multi-row data from a single form with scrolling;

7. Provide query-by-example forms;
8. Trap user-entered function and control keys;
9. Set up conditional screen attributes;
10. Access debugging tools;
11. Call 4GL or C library functions.

Creating applications with Informix-4GL is accomplished through an application development interface called the Programmer's Environment. Modules are created, compiled, and linked to the other modules of the application. This can also be accomplished with library functions. A variety of utility programs are included with the full Informix-4GL package.

The basis for automating the application development process comes from the use of terse keywords such as MENU, COMMAND, and HELP. Using these commands and their syntax, Informix-4GL accesses the organization's databases in its application program. The most important concept to note in using Informix-4GL to develop an application is that the data to be processed in the application is independent of the application. Because of this, different applications can access the same data using different views.

Creation of a ring menu can easily be accomplished using the keyword MENU. The user can make a selection by entering the first letter of the available options or by using the keyboard spacebar to move to the desired option and pressing return to confirm the request. Figure 4.5 illustrates a typical ring menu.

```
MAIN: Customer  Orders  Stock  Reports  Exit
Enter and maintain customer data
```

Figure 4.5 Ring Menu

The creation of the illustrated menu requires only a few lines of code using the Informix-4GL language. Figure 4.6 shows the code needed to create this menu. In the example the name of the menu appears after the keyword MENU. The options available to the user appear after the keyword COMMAND. A string following the command is listed on the second line of the screen below the menu options when that option is highlighted (see Figure 4.5).

```
MENU "MAIN"
COMMAND "Customer" "Enter and maintain customer data"
    CALL cust()
COMMAND "Orders" "Enter and maintain orders"
    CALL ord()
COMMAND "Stock" "Enter and maintain stock list"
    CALL stock()
COMMAND "Reports" "Print reports and mailing labels"
    CALL rept()
COMMAND "Exit" "Exit program and return to operating
                system"
```

Figure 4.6 MAIN Menu Routine

Below the COMMAND line are a series of steps that the program will execute for that particular option. In this case a function is called and when the function returns the menu is redrawn on the screen. Coding the same menu screen in a third generation language, C in this instance, requires about 44 lines of code.

Many other functions can be performed with a very few lines of code. In screens, default values can be designated and commands can be specified to automatically advance the cursor to the next field. Screens are created by writing a form specification file.

Other features that are of note are that Informix-4GL is case insensitive. In the examples presented the keywords were in uppercase for illustrative purposes only. Also, interfacing with routines written in the C programming language is possible when necessary. Since the language supports only addition, subtraction, multiplication, division, and exponentiation, a C routine would be required for any other calculation, such as returning a cosine value.

Informix-4GL is a programming language and therefore does not produce any code, in the traditional sense, automatically. Realization of the full power of the language requires the skill of a trained programmer/analyst.

The tax calculation model would be a good candidate for Informix-4GL. Databases can be developed for employees in the Informix-4GL environment with the 'CREATE DATABASE' command. A process to compute their gross pay and tax liability could be developed using the query language to manipulate the database. Reports could then be generated as necessary. As with COGEN, there is no analysis for completeness and consistency. Thus once again, the responsibility for good design rests with the programmer/analyst.

Informix-4GL is available for a variety of environments including a variety of micro-, mini-, and mainframe computers. Prices range from $995 for IBM PC type environments to $72,000 for IBM mainframe environments such as the Naval Postgraduate School's.

E. GENIFER

The final product is another from Bytel Corporation called GENIFER. With this software development tool the user can automatically create command files, or programs, using the dBASE III programming language from Ashton-Tate. Though GENIFER can alleviate many of the error-prone tasks that accompanies conventional programming methods, it does not relieve the programmer of the responsibility for good design. Before starting the task of developing an application program care must be taken in establishing database files and the relationships among them.

From the Main Menu shown in Figure 4.7, the user begins the development task. Application building with GENIFER is a three stage process. First, the specifications for the

```
MAIN MENU
 1. Data Dictionary
 2. Definitions
 3. Program Generators
 4. Customizer
 5. Help
 Q. Exit GENIFER
```

Figure 4.7 GENIFER Main Menu

database file must be entered via the data dictionary. Among the specifications are field names and types, picture templates, validation criteria, and aliases. Second, the menus, maintenance screens, and reports are defined. Finally, the actual dBASE III programs are generated.

For each field the data dictionary maintains the following:

1. Type--character, numeric, date, or logical;
2. Length;
3. Decimal;
4. Default value;
5. Validations--range, list, or file;
6. Picture--any acceptable dBASE III picture template without quotes.

The data dictionary does not support memo field types.

Selecting the definitions option the user defines menus, help screens, maintenance and inquiry screens, and report layouts for applications. Menus and help programs that are generated by GENIFER are dBASE III command files. Generated maintenance, inquiry, and report programs are procedures maintenance, inquiry, and report programs are procedures that are executed from menus.

Programs are generated automatically by GENIFER from user specifications utilizing the data dictionary and definitions specified. Programs consist of dBASE III source code and can be compiled using dBASE III compilers such as Clipper. Any sort of data processing that involves the manipulation of data from one or more databases can be programmed using GENIFER. The tax calculation model that has been cited previously is a good example.

If option 4 is selected from the Main Menu the user can customize Genifer for a particular environment. Among the environment options that must be customized for Genifer to execute properly is the specification of a text editor. The text editor is used to paint screens for menus, reports, and maintenance and inquiry. The text editor is called from within Genifer as needed by the user. Any text editor that returns an ASCII file can be used. Among the more common are WordStar, WordPerfect, and Edlin.

Using GENIFER, the FEDTAX example previously used was developed into a simple dBase III program. The programming task started with a design of the databases and the relationships between them. In this case, one database for the employees was created with fields for name, ID number, grosspay, and federal tax category. A second database with the fields federal tax category and federal tax rate was created. Genifer was used to create both databases. The first choice on the Genifer's Main Menu was selected and from there the process was a simple "fill in the blanks" operation. After the database was named, given an alias, and briefly described the fields were defined using an option called 'ZOOM' by Genifer. With this option the user is carried to the inner maintenance level of the Data Dictionary module to maintain the fields of the current database. The database is actually created with the 'INIT'(iate) option.

The next step in the programming process was to create a menu that listed the options available to the user. Once the menu is generated and the programs that support the options available are generated the programs are executed under dBASE III with the command "DO file-name", where file-name is the name selected for the generated program. For the example FEDTAX the menu has four choices, figure an employee's tax, update the employee database, update the federal tax database, and quit.

Option 2 from the Genifer Main Menu leads the user to another menu where the type of screen to be painted is designated. In addition to the menus and help screens option the user can also select maintenance and inquiry screens, reports, or quit. Like the data dictionary module, the definitions module is a two-tiered maintenance program. The top layer is used to define the screen and specify its parameters and the inner screen is used to paint the layout using the selected text editor. To define the menu screen it must be given a name, an exit key, a default select key, and the user will be asked if the screen is to be cleared before display. After definition, painting the screen is via the text editor. The user again selects the 'ZOOM' option to get to the editor. The only restriction here is that a 'field painting' character must appear somewhere on the screen. The character is defined in the Environment Customizer and the default value is the underscore.

Once the screen has been painted it must be defined. Definition indicates what action is to be taken when a user enters a particular selection from the menu. In the case of the tax example, selection 1, 2, or 3 will call dBASE procedures. Option 4, Quit, terminates the program and returns the user to the dBASE dot prompt.

To complete the programming exercise screens have to be painted and defined for maintenance of the two databases and the report. The procedure is basically the same for all with the report definition the most complex and powerful.

After the report screen has been defined and painted line assignments must be made. Each line can be designated in a variety of ways such as a report header that appears only on the first page of the report, a page header for each page, and so forth. The detail option is the selection that designates the actual contents of the report. It contains fields from the records in the databases or computed fields. For the tax example the name, id, grosspay, and federal tax fields were written directly and the state tax and total tax were computed. Designating each blank in a line of the report is done via another fill-in-the-blank menu that appears at the bottom of the screen during the report line assignment.

Generating the dBASE III code is done by Genifer through the use of skeleton files and the databases, definitions, and assignments made previously by the user. After selecting the program generation module the user is again taken to another menu to select either a menu and help screen program, maintenance or inquiry program, or a report program. Each of the options has a unique fill-in-the-blank screen that is completed by the user before Genifer creates the program. The menus and help screens are the simplest with no further action required other than selection of the menu program to be generated from the list of menu or help screens that have been defined. For maintenance programs the user has the option of generating programs that maintain more than one database.

In summary, program generation with Genifer is a somewhat repetitive task of moving through menus and filling in the blanks. Key to success, however, is again design. Genifer was not difficult to learn to use. Knowledge of dBASE III made it easier and it is probably not reasonable to expect it to eliminate the programmer from the user-programmer-application loop.

An additional feature of GENIFER is the security software distributed with the program. It allows installation of a password-protected user interface for the developed application with no additional programming. A user will be allowed four tries before termination. Successful access requires that the user provide an ID that matches a name in the user's database file and must then match the password specified for that user. A successful login will allow access to specified menus.

GENIFER comes with a tutorial disk and is not copy-protected. Although the system can be run on a minimum two floppy disk configuration, a hard disk is recommended. GENIFER is priced at $395.

F. FEATURES OF SYSTEMS

The systems discussed can all be evaluated by the features presented in the previous chapter. Table 1 lists the features applicable for the systems discussed in this chapter. The dominant input feature of these systems is VHLL. This supports Houghton's observation that automatic programming systems display this trend. The most common output feature remains listings with VHLL gaining in prominence. As 4GLs mature VHLL will most likely continue to display growth.

TABLE 4.1

SYSTEM FEATURES

System Name	Features		
	Input	Function	Output
USE.IT	code VHLL	formatting translation synthesis cross ref. error check interface analysis simulation	listings graphics user text source code data
COGEN	code VHLL	formatting synthesis error check	listings source code
INFORMIX-4GL	code VHLL	synthesis error check	listings VHLL
GENIFER	VHLL	formatting synthesis error check	listings VHLL

There may be other features that could be identified through dynamic analysis and benchmarking. Resource constraints prevented such study. As a result of the static nature of the reviews, some desired information was unavailable due to its proprietary nature.

G. SUMMARY

This chapter has presented a review of four software development tools that are available from industry sources. Appendix A is a listing of other software development systems that are classified with the systems reviewed as Automatic Code Generators. Two of them, USE.IT and COGEN, produce applications in third generation languages.

INFORMIX-4GL and GENIFER work with the more recent fourth generation languages. All of the systems interface with the programmer through menus prompting for inputs. None of the products purport to eliminate the programmer from the user-programmer-application link that was presented in Chapter II. With the exception of USE.IT, all of the systems are available to the microcomputer user.

Comparing the four systems briefly it can be seen that USE.IT displays the most potential for accommodating a wide variety of applications. It is also the most costly and suffers from a fairly long learning curve according to interviews with two users [Refs. 16,17]. USE.IT along with COGEN and INFORMIX-4GL seem to be tools geared more to the larger production environments. GENIFER would be considered more of an individual productivity enhancement tool. A decision by management to make one of these tools a standard for development would require an in-depth cost-benefit analysis prior to capital investment. USE.IT with its complex, long learning curve and relatively high cost would seem most attractive to a large production software environment or one where a specific software language, say ADA perhaps, environment has been designated. On the other hand, COGEN is attractive to many software development centers where the COBOL language is in use. A larger business that has yet to develop a standard language tool may be inclined to select the INFORMIX-4GL environment and gain the flexibility that accompanies the separation of data from applications. Finally, GENIFER will be attractive to a production center in the microcomputer environment. Individual users as well as application development centers will find GENIFER an attractive option.

The following chapter presents the conclusions and recommendations based on the findings of this study.

V. CONCLUSIONS AND RECOMMENDATIONS

A. INTRODUCTION

This thesis has discussed the important role of software in a computer system, the development of software, and the software crisis faced by the information systems manager. Among the potential solutions to the software crisis is the automation of some or all parts of the SDLC. This thesis has concentrated on the software development tools that have automated the coding stage of the SDLC. Chapter III presented a taxonomy of software development tools which was developed by the National Bureau of Standards. The features identified in that study were compared to those of four commercially available automatic code generators in Chapter IV.

B. CONCLUSIONS

From the data gathered in this thesis explicit answers to the thesis questions can be presented. First, it appears that most products available concentrate on the automation of the code writing stage of the SDLC. USE.IT, however, takes a more ambitious approach by attempting automation of the entire life cycle. Although COBOL code generating systems are more widespread, the fourth generation language application development systems are rapidly gaining prominence.

The primary reason for considering automatic code generators is to relieve the burden on the programmer and thereby relieve the software crisis being experienced by an organization and if possible to eliminate the programmer altogether. Generally, the systems considered by this thesis can contribute partially to this end. The U.S. Army's First Recruiting Brigade at Fort Meade, MD increased their programmer productivity 800% with GENIFER [Ref. 14:p. 46]. This more than meets the criteria set forth by Martin to define a fourth generation language. While this is a startling improvement it did not eliminate the need for a programmer in the application development. None of the systems reviewed claimed that the programmer could be eliminated and have application development become exclusively a user responsibility. The use of 4GLs and application development systems have moved the industry closer to that position. Figure 5.1 illustrates a relative relationship between the software systems reviewed in this thesis and their user audience. It verifies that the fourth generation type tools are closer to eliminating the programmer but it also verifies that the programmer still has a large role to play in the development of software applications.

There is a learning curve associated with using all of these programs and a manager must decide whether the users can effectively and efficiently develop their own

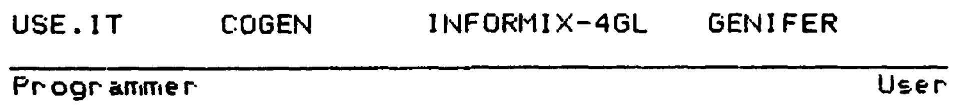

Figure 5.1 Programmer, User, Software Perspective

applications. The pressure to decide in favor of user developed applications will mount as the software crisis deepens.

All of the products that claim to produce error-free code do in fact write code that is ready to be compiled and executed. All programs require that the user provide a good design of the system to the generator to produce the code. Through the use of menus and an interactive environment the programmer provides the information to the development tool that is usually used as input to some sort of skeleton file to write the code.

Among the benefits that can be realized from the use of automatic code generating systems is standardization. Applications developed will all be written in the same style and those that document will provide standardized documentation. With personnel turnover a fact of life, the training of replacement personnel will be standardized also.

Lower cost of development is another of the benefits to be realized with automated code generating systems. A shorter SDLC will reduce development personnel costs and deliver software to the user sooner. This quicker turnaround from requirement to application brings the benefits of the program into use sooner and contributes to a shorter payback period for the investment in the application. Additionally, maintenance becomes a task where savings can be realized.

Disadvantages are also associated with the use of automatic code generators. Many have an initial capital outlay that is significant. A shortcoming that is evident in the COGEN system is that it is designed for use in the development of business applications. A system that can develop applications for any type of problem is desirable. Finally, another disadvantage associated with automatic code generators is the learning curve associated with them. USE.IT is a complex system that requires a skilled programmer/analyst to realize its full potential. GENIFER is not nearly as complex but neither is it as potentially powerful.

In summary, the automatic code generators have potential to relieve the software crisis to some degree. They are not, however, the panacea.

C. RECOMMENDATIONS

The use of automatic code generating systems can relieve an organization's software crisis. Before moving to put this tool into inventory, however, a complete cost/benefit analysis must be performed. A tool must then be fully utilized to realize the available benefits.

Fourth generation languages and associated programming tools are the level of technology that should be sought by organizations, particularly smalle:, newer organizations. Establishing databases independent of the applications may help the organizations avoid the software crisis to some extent. The lack of a fourth generation language standard may deter some organizations unnecessarily. A sound evaluation by management and commitment to the chosen product will establish the product as the standard within the organization.

Further study areas related to this thesis include dynamic evaluation of the tools and benchmarking. A dynamic evaluation would help the user determine all features, length of the learning curves, limitations, and other items. A particularly interesting determination would be the level of sophistication needed by a user to develop applications using these tools thereby eliminating the programmer from the development loop. A benchmark test would provide measures to compare the different tools to one another. One possible benchmark test would be to determine the efficiency of the code the system produced.

It is clear that automation of software development must play a significant role in future system development. The length of the list of systems available in the Appendix supports this statement and indicates that the future holds many more such systems.

APPENDIX

AUTOMATIC CODE GENERATING SYSTEMS

This appendix lists additional software systems that fall into the categories of automatic code generating software examined in this thesis. The system title, publisher, business address and telephone numbers are listed.

STRUCTURES

Ken Orr and Associates
1725 Gage Boulevard
Topeka, KS 66604-3379
913-273-0653
800-255-2459

THE GENERATOR

Computer Technology, Inc.
11101 NE 8th
Belevue, WA 98004
206-455-1700

CODE SHELL/COBOL

Morrison-Rooney Associates, Ltd.
910 S. Michigan Ave., Suite 520
Chicago, IL 60605
312-922-5980

CL-1

Software Design Associates, Inc.
71 Fifth Ave.
New York, NY 212-741-5200

FASTBASE FASTBASE III

Fourcolor Data Systems, Inc.
7011 Malabar St.
Dayton, OH 45459
513-433-3780

CRT (COBOL Reprogramming Tool)

Cybernetics, Inc.
8041 Newman Ave., Suite 208
Huntington Beach, CA 92647
714-848-1922

QUICKCODE

Fox and Geller, Inc.
604 Market St.
Elmwood Park, NJ 07407
201-794-8883

PEARL SD PROGRAM GENERATOR

Pearlsoft, Inc.
25195 Southwest Parkway
P.O. Box 638
Wilsonville, OR 97070
503-682-3636

DATA MASTER 1 REV 3

Applied Micro Business Systems, Inc.
177-F Riverside Ave.
Newport Beach, CA 92663
714-759-0582

BUSIGEN/CS PROXI/PROXI COBOL PROGRAM GENERATOR

Data General, Corp.
4400 Computer Drive
Westboro, MA 01581
617-366-8911

EZ PROG

New England Systems Technology, Inc.
226 South St.
Hopkinton, MA 01748
617-435-9031

FORCE

Point 4 Data Corp.
2569 McCabe Way
Irvine, CA 92714
714-863-1111

COBOL PROGRAM GENERATOR/INFORMATION SUPPORT SYSTEM
David R. Black and Associates, Inc.
P.O. Box 44146
Pittsburgh, PA 15205
412-787-5100

AMS AUTOMATIC TEST PROGRAM GENERATOR

AMS--Advanced Microsolutions
1100 Alma St.
Menlo Park, CA 94025
415-325-7694

THE BALER

Brubaker and Associates, Inc.
116 W. Main St.
P.O. Box 511
Delphi, IN 46923
317-564-2584

COBFORMAT

Barratt Edwards International Corp.
2921 Eastlake Ave. E.
Seattle, WA 98102
206-325-1011

FASTBALL--76

Brown Bros. Enterprises
8079 Wabasis Ave., NE
Rockford, MI 49341
616-691-7193

HIBOL

Delphi Data Systems, Inc.
9615 Girard Ave. S.
Bloomington, MN 55431
612-881-4666
800-328-4827

KOPE

KOS and Associates, Inc.
McKnight Park Dr.
Suite 203
Pittsburgh, PA 15237
412-367-7444

PROMACS/CICS

Management and Computer Services, Inc.
Great Valley Corporate Center
P.O. Box 826
Valley Forge, PA 19482
215-648-0730

GENPREP-1

Miroda Software Corp.
P.O. Box 10089
Chicago, IL 60610
312-743-2755

PRO-2

Prodata, Inc.
4477 Emerald
Suite C-100
Boise, ID 83706
208-342-6878

GTP

Allen, Emerson Franklin, Inc.
P.O. Box 928
Katy, TX 77449
713-391-8570

CLARION

Barrington Systems, Inc.
150 East Sample Road
Pompano Beach, FL 33064
305-785-4555
800-354-5444

PRO-IV

Honeywell Inc.
Information Technology Center
120 Howard Street
San Francisco, CA 94105
415-974-4304

CLINE/CENGLISH

Cline, Inc.
3550 Camino del Rio North
San Diego, CA 92108
800-544-1104

EXCELERATOR

Index Technology Corp.
5 Cambridge Center
Cambridge, MA 02142

BASIS

Information Dimensions, Inc.
655 Metro Place South
Dublin, OH 43017-1396
800-328-2648

NOMAD2

D and B Computing Services
187 Danbury Rd.
Wilton, CT 06897
203-762-2511

CONSENSUS

Martin-Marietta Data Systems
CONSENSUS INFO
P.O. Box 2392
Princeton, NJ 08540
800-257-5171

MANTIS
CINCOM
2300 Montana Avenue
P.O. Box 11189
Cincinnati, OH 45211
513-662-2300

LIST OF REFERENCES

1. Boehm, Barry W., "Software and Its Impact: A Quantitative Assessment," Datamation, Vol. 19, p. 23, May 1973.

2. Lin, Herbert, "The Development of Software for Ballistic Missile Defense," Scientific American, Vol. 253, p. 56, December 1985.

3. Kroenke, David M., Business Computer Systems, p. 25, Mitchell Publishing Inc., 1984.

4. Hamilton, M. and Zeldin, S., "The Functional Life Cycle Model and Its Automation: USE.IT," The Journal of Systems and Software, Vol. 3, pp. 25-61, Marc 1983.

5. Keen, Peter G.W. and Scott Morton, Michael S., Decision Support Systems: An Organizational Perspective, pp. 11-12, Addison-Wesley Publishing Company, Inc., 1978.

6. Pressman, Roger S., Software Engineering: A Practitioner's Approach, p. 289, McGraw-Hill Book Co., 1982.

7. Brooks, Jr., Frederick P., "The Mythical Man-Month," Datamation, Vol. 20, p. 35, December 1974.

8. Gifford, David and Spector, Alfred, "The TWA Reservation System," Communications of the ACM, Vol. 27, p. 651, July 1984.

9. Martin, James and Hershey, Al, "Software Engineering Depends on Information Engineering," Software News, Vol. 5, p. 60, March 1986.

10. Martin, James, Application Development Without Programmers, p. 4, Prentice-Hall, Inc., 1982.

11. Robinson, Philip and Edwards, Jon R., "The Atari 1040ST," BYTE, Vol. 11, p. 87, March 1986.

12. Sippl, Roger J., "Marketing Trends Launch a New Era for Software Developers," Hardcopy, Vol. 6, p. 44, June 1986.

13. Malvino, Albert P., Digital Computer Electronics, Vol. 2, p. 145, McGraw-Hill Book Co., 1983.

14. "Brigade Tackles Backlog with Application Generator," Government Computer News, Vol. 5, 4 July 1986.

15. Personal interview between Dan Pines, Bytel Corporation, Berkeley, CA and the author, 17 September 1986.

16. Telephone interview between Shirley Jahn, Navy Management Systems Support Office, and the author, 15 June 1986.

17. Telephone interview between Frank T. Lawrence, U.S. Army Logistics Center, and the author, 16 June 1986.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Director, Information Systems (OP-945) Office of the Chief of Naval Operations Navy Department Washington, D.C. 20350-2000	1
4.	Chairman, Computer Technology Programs Code 37 Naval Postgraduate School Monterey, California 93943-5000	1
5.	Professor Dan K. Dolk, Code 54Dk Department of Administrative Sciences Naval Postgraduate School Monterey, California 93943-5000	2
6.	LCDR Sherman L. O'Brien, USN 710 Winfred Dr. N. Orange Park, Florida 32073	2